Using Constraint Satisfaction To Improve Deterministic 3-SAT

Konstantin Kutzkov
IT University of Copenhagen, Denmark
kutzkov@googlemail.com

Dominik Scheder
ETH Zürich
dscheder@inf.ethz.ch

July 27, 2010

Abstract

We show how one can use certain deterministic algorithms for higher-value constraint satisfaction problems (CSPs) to speed up deterministic local search for 3-SAT. This way, we improve the deterministic worst-case running time for 3-SAT to $O(1.439^n)$.

1 Introduction

Among NP-complete problems, boolean satisfiability, short SAT, is perhaps the most intensively studied, rivaled only by graph colorability. Within SAT, the case of k-SAT, in which the input is restricted to formulas containing only clauses of size at most k, has drawn most attention, in particular the case $k = 3$. The currently best algorithms for 3-SAT are based on two rather different ideas. The first is local search. In [8], Schöning gives an extremely simple randomized local search algorithm, and its running time of $O^*((4/3)^n)$ is still close to the best known running times (here, n is the number of variables, and we use the notation O^* to suppress factors that are polynomial in n). The second idea is to process the variables of F in random order, assigning them 0 or 1 randomly, unless one choice is "obviously wrong". This was introduced by Paturi, Pudlák, and Zane [4], achieving a running time of $O^*(1.59^n)$. By using a less obvious notion of "obviously wrong" (and a much more complicated analysis), Paturi, Pudlák, Saks, and Zane [3] significantly improved this, however not beating Schöning's bound of $(4/3)^n$. Iwama and Tamaki combined these two approaches to obtain a running time of $O^*(1.3238^n)$. Rolf [5] improved the analysis of that algorithm and showed that its running time is $O^*(1.32216^n)$, the currently best bound.

Deterministic algorithms for 3-SAT do not achieve these running times. The currently best deterministic algorithms can all be seen as attempts to derandomize Schöning's local search algorithm. The first attempt is by Dantsin et al. [2] and is based on a simple recursive local search algorithm, combined with a construction of covering codes. This achieves a running time of $O^*(1.5^n)$.Dantsin et al. also show how the recursive algorithm can be improved to achieve an overall running time of $O^*(1.481^n)$. Subsequent papers (Brueggeman and Kern [1], Scheder [6]) improve the local search algorithm by giving more sophisticated branchings. This paper also improves the running time by improving the recursive local search algorithm, but is still qualitatively different from previous work: We show that, under certain circumstances, one can translate those 3-CNF formulas that

constitute the worst-case for previous algorithms into a constraint satisfaction problem (CSP) with more than two values (boolean satisfiability problems are CSPs with two values), which can be solved quite efficiently.

Notation

A CNF formula F is a conjunction (AND) of clauses. A clause is a disjunction (OR) of literals, and a literal is either a boolean variable x or its negation $\bar{x}$. A k-clause is a clause with exactly k distinct literals. For example, $(x \vee \bar{y} \vee \bar{z})$ is a typical 3-clause. A k-CNF formula is a CNF formula consisting of k-clauses. An *assignment* is a mapping of variables to truth values. We use the numbers 0 and 1 to denote `false` and `true`. If V is a set of n variables, an assignment to V can be seen as a bit string of length n, i.e., an element of $\{0,1\}^n$. For two such assignments α, β, we denote by $d_H(\alpha, \beta)$ their *Hamming distance*, i.e., the number of variables on which they disagree.

1.1 Randomized Local Search

We will briefly describe Schöning's local search algorithm [8]. Let F be a 3-CNF formula and let α be some truth assignment to the variables of F. If α does not satisfy F, the algorithm arbitrarily picks a clause C that is unsatisfied by α. For simplicity, suppose $C = (\bar{x} \vee \bar{y} \vee \bar{z})$. The algorithm uniformly at random picks a variable from C and flips the value α assigns to it. This is repeated $3n$ times. If the algorithm did not find a satisfying assignment within $3n$ steps, it gives up. Schöning [8] proved the following lemma.

Lemma 1 (Schöning [8])**.** *Suppoe F is a satisfiable 3-CNF formula and α^* is a satisfying assignment. Let α be a truth assignment and set $r := d_H(\alpha, \alpha^*)$. Then the above algorithm finds a satisfying assignment with probability at least $(1/2)^r$.*

Schöning now picks the initial assignment α uniformly at random from all truth assignments and then starts the local search we described above. For a fixed satisfying assignment α^*, we observe that

$$\Pr[d_H(\alpha, \alpha^*) = r] = \frac{\binom{n}{r}}{2^n} ,$$

and therefore overall success probability is at least

$$\sum_{r=0}^{n} \frac{\binom{n}{r}}{2^n} \left(\frac{1}{2}\right)^r = \left(\frac{3}{4}\right)^n .$$

By repetition, this yields a Monte Carlo algorithm of running time $O^*((4/3)^n)$. We see that Schöning's algorithm uses randomness in two ways: First to choose the initial assignment, and second to steer the local search. It turns out that one can derandomize the first random choice at almost no cost (the running time grows by a polynomial factor in n). Derandomizing local search itself, however, is much more difficult, and all currently known versions yield a running time that is exponentially worse than Schöning's randomized running time.

1.2 Deterministic Local Search

We sketch the result Dantsin et al. [2], who were the first to give a deterministic algorithm based on local search, which runs in time $O^*(1.5^n)$. For that, consider the parametrized problem Ball-3-SAT.

Ball-3-SAT: Given a 3-CNF formula F over n variables, a truth assignment α to these variables, and a natural number r. Is there an assignment α^* satisfying F such that $d_H(\alpha, \alpha^*) \le r$?

We call this problem Ball-3-SAT because it asks whether the Hamming ball of radius r around α, denoted by $B_r(\alpha)$, contains a satisfying assignment. The merits of [2] are twofold. First, they give a simple recursive deterministic algorithm solving Ball-3-SAT in time $O^*(3^r)$; If α does not satisfy F, pick an unsatisfied clause C. There are $|C| \le 3$ ways to locally change α as to satisfy C. Recurse on each. One can regard this recursive algorithm as a derandomization of Schöning's local search algorithm. It comes at a cost, however: Its running time is $O^*(3^r)$, whereas Schöning's local search has a success probability of $(1/2)^r$. Therefore, a "perfect" derandomization should have running time $O^*(2^r)$. Second, they show that an algorithm $\mathcal{A}$ solving Ball-3-SAT in time $O^*(a^r)$ yields an algorithm $\mathcal{B}$ solving 3-SAT in time

$$O^*\left(\left(\frac{2a}{a+1}\right)^n\right) , \tag{1}$$

and furthermore, $\mathcal{B}$ is deterministic if $\mathcal{A}$ is. This works by covering the set of satisfying assignments with Hamming balls of radius r and solving Ball-3-SAT for each ball. Formally, one constructs a *covering code* $\mathcal{C}$ of radius r, which is a set $\mathcal{C} \subseteq \{0,1\}^n$ such that

$$\bigcup_{\alpha \in \mathcal{C}} B_r(\alpha) = \{0,1\}^n$$

and then solves Ball-3-SAT for each $\alpha \in \mathcal{C}$.

Can one solve Ball-3-SAT *deterministically* in time $O^*(2^r)$? Nobody has achieved that yet, although a lot of progress has been made. By devising clever branching rules (and proving non-trivial lemmas), one can reduce the running time to $O^*(a^r)$ for $a < 3$. Dantsin et al. already reduce it to $O^*(2.848^r)$, Brueggeman and Kern [1] to $O^*(2.792^r)$, and Scheder [6] to $O^*(2.733^r)$. The approach which we present here is different. Instead of designing new branching rules, we transform worst-case instances of Ball-3-SAT into $(3,3)$-CSP formulas over r variables, which one can solve more efficiently. Of course, in reality several subtleties arise, and the algorithm becomes somewhat technical. Still, we achieve a substantially better running time:

Theorem 2. Ball-3-SAT *can be solved deterministically in time $O^*(a^r)$, where $a \approx 2.562$ is the largest root of $x^2 - x - 4$. Together with (1), this gives a deterministic algorithm solving 3-SAT in time $O^*(1.439^n)$.*

2 An Improved Algorithm for Ball-3-SAT

In this section we will describe an algorithm for Ball-3-SAT. At first, our algorithm for Ball-3-SAT is not much different from the one in Dantsin et al. For simplicity assume $\alpha = (1, \ldots, 1)$, and we want to decide whether there is a satisfying assignment α^* that sets at most r variables to 0. We will describe a recursive algorithm. By $L(r)$ we denote the number of leaves of its recursion tree.

Intersecting Unsatisfied Clauses

Suppose F contains two negative 3-clauses that intersect in one literal, for example $(\bar{x} \vee \bar{y} \vee \bar{z})$ and $(\bar{x} \vee \bar{u} \vee \bar{v})$. The algorithm has one "cheap" choice, namely setting x to 0, and

four "expensive" choices, namely setting one of y, z and one of u, v to 0. Recursing on all five possibilities yields the recurrence

$$L(r) \leq L(r-1) + 4L(r-2) \ .$$

Standard methods show that this recurrence has a solution in $O(a^r)$, with $a \approx 2.562$ being the largest root of $x^2 - x - 4$. If F contains two negative 3-clauses intersecting in two literals, we obtain an even better recurrence: Let $(\bar{x} \vee \bar{y} \vee \bar{z})$ and $(\bar{x} \vee \bar{y} \vee \bar{u})$ be those two clauses. The algorithm has two cheap choices, namely setting x to 0 or y to 0. Besides this, it has one expensive choice, setting z *and* u to 0. This gives the following recurrence:

$$L(r) \leq 2L(r-1) + L(r-2) \ .$$

This recurrence has a solution in $O\left(\left(\sqrt{2}+1\right)^r\right) \leq O(2.415^r)$.

2.1 Disjoint Unsatisfied 3-Clauses

Let $\mathrm{Neg}(F)$ denote the set of negative clauses in F, i.e., clauses with only negative literals. Above we showed how to handle the case in which $\mathrm{Neg}(F)$ contains intersecting clauses. From now on, unless stated otherwise, we will assume that $\mathrm{Neg}(F)$ consists of pairwise disjoint negative 3-clauses. In all previous improvements to deterministic local search, the case where $\mathrm{Neg}(F)$ consists of r pairwise disjoint 3-clauses constitutes the worst case. Somewhat surprisingly, we can solve this case rather quickly, using a deterministic algorithm for $(3,3)$-CSP by Scheder [7]. We call an assignment *exact* if it sets exactly $|\mathrm{Neg}(F)|$ variables to 0, namely exactly one in each clause in $|\mathrm{Neg}(F)|$. Here are two simple observations: (i) if $|\mathrm{Neg}(F)| > r$, then $B_r(1,\ldots,1)$ contains no satisfying assignment; (ii) if $|\mathrm{Neg}(F)| = r$, then every satisfying assignment in $B_r(1,\ldots,1)$ is exact.

Lemma 3. *Suppose* $\mathrm{Neg}(F)$ *consists of* r *pairwise disjoint* 3*-clauses. Then* Ball-3-SAT *can be solved in time* $O^*(2.077^r)$

Proof. If $\mathrm{Neg}(F)$ consists of r pairwise disjoint 3-clauses, then F has a satisfying assignment α^* in $B_r(1,\ldots,1)$ if and only if it has an exact satisfying assignment. An exact assignment can satisfy each $C \in \mathrm{Neg}(F)$ in three different ways: Through its first, second, or third literal. We introduce a ternary variable x_C to represent these three choices. For example, if $C = (\bar{x} \vee \bar{y} \vee \bar{z})$, then every occurrence of the literal y can be replaced by the literal $(x_C \neq 2)$, and a literal $\bar{y}$ can be replaced by $(x_C \neq 1 \wedge x_C \neq 3)$. If $u \in \mathrm{vbl}(F) \setminus \mathrm{vbl}(\mathrm{Neg}(F))$, we can replace the literal u by `true` and $\bar{u}$ by `false`, as every exact assignment sets u to 1. In this manner, we translate an instance of Ball-3-SAT into a $(3,3)$-CSP problem over r variables. By a result of [7], one can solve this in time $O^*(2.077^r)$. ∎

Let $m := |\mathrm{Neg}(F)|$. We have just seen that the case $m = r$ is relatively easy. If $m < r$ then we have a "surplus budget" of $r - m$ that we can spend on satisfying multiple literals in some $C \in \mathrm{Neg}(F)$, or setting variables u to 0 that do not occur in $\mathrm{Neg}(F)$ at all. This will make things more complicated and lead to a worse running time:

Lemma 4. *Suppose* $\mathrm{Neg}(F)$ *consists of pairwise disjoint* 3*-clauses. Then* Ball-3-SAT *can be solved in time* $O^*(b^r)$*, where*

$$b = \frac{(5+\sqrt{57})^2}{4(8+\sqrt{57})} \approx 2.533 \ .$$

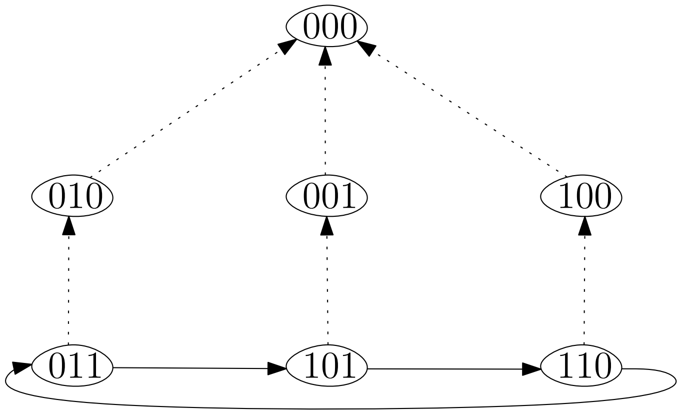

Figure 1: A graph on the seven colors with two kinds of edges. The seven colors represent the seven ways to satisfy a clause $(\bar{x} \vee \bar{y} \vee \bar{z})$.

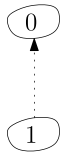

Figure 2: A graph on the two truth values $\{0, 1\}$, representing the two ways to set a variable $x \in V'$.

In the rest of this section we will prove the lemma. Write $V' := \mathrm{vbl}(F) \setminus \mathrm{vbl}(\mathrm{Neg}(F))$. Let $C = (\bar{x} \vee \bar{y} \vee \bar{z}) \in \mathrm{Neg}(F)$. An assignment to the variables x, y, z can be represented by a string in $\{0, 1\}^3$. Seven of them satisfy C: 011, 101, 110, 001, 010, 100, and 000. We call them *colors*. The first three colors are *exact*, the latter four *dirty*. Any assignment satisfying $\mathrm{Neg}(F)$ induces a 7-coloring of $\mathrm{Neg}(F)$. An assignment is exact if and only if it sets every $u \in V'$ to 1 and assigns every negative clause an exact color (i.e., 011, 101, or 110). We define a graph G on the seven colors, see Figure 1. This graph has two types of edges: solid and dotted ones.

Definition 5. *For two colors c, c', let $d(c, c')$ be the minimum number of solid edges on a directed path from c to c' (and ∞ if no such path exists), and* $\mathrm{cost}(c, c')$ *the minimum number of dashed edges (∞ if no such path exists).*

For example $d(011, 100) = 2$ and $\mathrm{cost}(011, 100) = 1$, but $d(010, 011) = \mathrm{cost}(010, 011) = \infty$. Let α, β be two assignments that satisfy $\mathrm{Neg}(F)$. Recall that α and β induce a 7-coloring of $\mathrm{Neg}(F)$, thus for $C \in \mathrm{Neg}(F)$, we write $\alpha(C), \beta(C)$ to denote this color. To a variable $x \in V'$, the assignment α does not assign one of the seven colors, but simply a truth value, 0 or 1. To simplify notation, we define $d(0, 1) = d(1, 0) = 0$, $\mathrm{cost}(1, 0) = 1$, and $\mathrm{cost}(0, 1) = \infty$. Just think of a graph on vertex set $\{0, 1\}$ with a dotted edge from 1 to 0, and no edge from 0 to 1 (Figure 2).

We define a "horizontal distance" from α to β

$$d(\alpha, \beta) := \sum_{C \in \mathrm{Neg}(F)} d(\alpha(C), \beta(C)) \ . \tag{2}$$

and a "vertical distance" from α to β

$$\mathrm{cost}(\alpha, \beta) := \sum_{C \in \mathrm{Neg}(F)} \mathrm{cost}(\alpha(C), \beta(C)) + \sum_{x \in V'} \mathrm{cost}(\alpha(x), \beta(x)) \ . \tag{3}$$

We say *horizontal* and *vertical* because these terms correspond to the orientation of the two types of edges in Figure 1, as the solid edges are horizontal and the dotted edges are vertical.

Proposition 6. *Let α, β be two assignments satisfying* $\mathrm{Neg}(F)$.

1. *If α, β are exact, then $d(\alpha, \beta) \le r$ and* $\mathrm{cost}(\alpha, \beta) = 0$.
2. *If α is exact, then $d(\alpha, \beta) \le r$ and* $\mathrm{cost}(\alpha, \beta)$ *is finite.*
3. *If $B_r(1, \dots, 1)$ contains an assignment α^* satisfying F, then there is an exact assignment β such that $d(\beta, \alpha^*) = 0$ and* $\mathrm{cost}(\beta, \alpha^*) \le r - m$.

We want to mimic the $(3,3)$-CSP of [7], although in the case $|\mathrm{Neg}(F)| < r$, we cannot directly translate our instance of BALL-3-SAT into an instance of $(3,3)$-CSP. The idea, roughly speaking, is to imitate the $(3,3)$-CSP-algorithm on the exact colors, while using a traditional branching algorithm to search through dirty colors. Let us be more precise:

We cover the set of exact assignments by a good covering code $\mathcal{C}$. Formally, we want a (small) set $\mathcal{C}$ of exact assignments such that for every exact β, there is some $\alpha \in \mathcal{C}$ such that $d(\alpha, \beta) \le s$, where $s \in \mathbb{N}$ is a suitably chosen integer.

Proposition 7. *Let $\mathcal{C}$ as described. Then for every satisfying assignment $\alpha^* \in B_r(1, \dots, 1)$, there exists an exact assignment $\alpha \in \mathcal{C}$ such that $d(\alpha, \alpha^*) \le s$ and* $\mathrm{cost}(\alpha, \alpha^*) \le r - m$.

Proof. By point (3) of Proposition 6, there is an exact assignment β such that $d(\beta, \alpha^*) = 0$ and $\mathrm{cost}(\beta, \alpha^*) \le r - m$. By the properties of $\mathcal{C}$, there is an $\alpha \in \mathcal{C}$ such that $d(\alpha, \beta) \le s$. By point (1) of Proposition 6, $\mathrm{cost}(\alpha, \beta) = 0$. Since d and cost obey the triangle inequality (which is easy to verify), it follows that $d(\alpha, \alpha^*) \le s$ and $d(\alpha, \alpha^*) \le r - m$. □

The main idea behind the algorithm of Dantsin et al. was to focus not on 3-SAT itself but on the parametrized problem BALL-3-SAT. We will do the same here. We define the following decision problem:

Double-Ball-SAT. Given a 3-CNF formula with $m \le r$ pairwise disjoint negative clauses, an assignment α satisfying $\mathrm{Neg}(F)$, and two integers s and t: Is there an assignment α^* satisfying F such that $d(\alpha, \alpha^*) \le s$ and $\mathrm{cost}(\alpha, \alpha^*) \le t$?

There are two special cases in which DOUBLE-BALL-SAT can be solved rather quickly. First, consider the case $t = 0$. If C is a clause unsatisfied by α, we have three possibilities to change α: At the first, second, or third literal of C. However, we change α only along solid (horizontal) edges, since $t = 0$ anyway rules out any assignment α^* differing from α by a dotted (horizontal) edge. Since every vertex in G is left by at most one solid edge, the running time is $O^*(3^s)$. This algorithm is in fact identical to the deterministic local search algorithm for BALL-$(3,3)$-CSP in [7].

Second, consider the case $s = 0$. If α does not satisfy clause C, we again have three possibilities to change α. This time, however, solid edges are ruled out by $s = 0$. Since

every vertex (color) is left by at most one dotted edge, this yields a running time of $O^*(3^t)$. In fact, the running time is better, namely $O^*(2^t)$: Let C be a clause not satisfied by α. Clearly $C \notin \mathrm{Neg}(F)$, since α satisfies $\mathrm{Neg}(F)$. Hence C has at least one positive literal x. But look at Figures 1 and 2: Following a dotted edge always means setting one additional variable to 0, never setting one to 1. Therefore, the positive literal x cannot become satisfied by following a dotted edge, and there are actually at most two choices to change α, resulting in a running time of $O^*(2^t)$.

Let us summarize: The problem Double-Ball-SAT with parameters s and t can be solved in time $O^*(2.077^s)$ if $t = 0$ and $O^*(2^t)$ if $s = 0$. It would be nice if these two border cases combined into a general running time of $O^*(2.077^s 2^t)$. Alas, this is not true. Or rather we do not know how.

An Algorithm for Double-Ball-SAT

We give a recursive algorithm `double-ball-search`(F, α, s, t) for Double-Ball-SAT. We start with an assignment α satisfying $\mathrm{Neg}(F)$. As long as α does not satisfy F, and $s, t \geq 0$, we modify α locally, in the hope of coming closer to a satisfying assignment, and continue recursively.

There are two simple base cases. First, if $s < 0$ or $t < 0$, the algorithm returns `failure`. If $s, t \geq 0$ and α satisfies F, it returns α. Otherwise, $s, t \geq 0$, and there is some clause $C \in F \setminus \mathrm{Neg}(F)$ which α does not satisfy (recall that α satisfies $\mathrm{Neg}(F)$). The clause C has at most three literals, and at most two of them are negative. For each literal $\ell \in C$, the algorithm modifies α at one position in order to satisfy ℓ. Note that each color has at most two outgoing edges, so to satisfy ℓ, there are at most two direct ways to change the value of ℓ under α. This means that each literal entails at most two recursive calls. The exact nature of these calls depends on the literal itself.

We investigate which recursive calls are necessary when we try to change the value α assigns to $\ell \in C$. Note that α currently does not satisfy ℓ. There are two cases: Either $\mathrm{vbl}(\ell) \in \mathrm{vbl}(\mathrm{Neg}(F))$ or $\mathrm{vbl}(\ell) \in V'$ (above, we defined V' to be $\mathrm{vbl}(F) \setminus \mathrm{vbl}(\mathrm{Neg}(F))$). We start with the less interesting case.

Case 1. $\mathrm{vbl}(\ell) \in V'$.

Case 1.1 ℓ is a positive literal, i.e., $\ell = x$ for some $x \in V'$. In this case, $\alpha(x) = 0$, and for x, no edge leads back from 0 to 1. The algorithm gives up in this branch.

Case 1.2 ℓ is a negative literal, i.e., $\ell = \bar{x}$ for some $x \in V'$. In this case, $\alpha(x) = 1$, and there is only one way to change α: Take the dotted edge, setting x to 0. The algorithm takes one recursive call:

$$\texttt{double-ball-search}(F, \alpha[x \to 0], s, t-1) \ .$$

Case 2. $\mathrm{vbl}(\ell) \in \mathrm{vbl}(\mathrm{Neg}(F))$. In this case, there is exactly one clause $D = (\bar{x} \vee \bar{y} \vee \bar{z}) \in \mathrm{Neg}(F)$ such that $\ell \in \{x, y, z, \bar{x}, \bar{y}, \bar{z}\}$.

Case 2.1 Suppose $\alpha(D)$ is dirty. There is only one outgoing edge, which is dotted, leading to some assignment α'. Hence there is at most one recursive call, regardless of

the literal ℓ:

$$\texttt{double-ball-search}(F, \alpha', s, t-1) \ .$$

In the remaining cases, we can assume that $\alpha(D)$ is pure. Without loss of generality, we assume that $\alpha(D) = 011$. This means that $\alpha(x) = 0$ and $\alpha(y) = \alpha(z) = 1$. Since α does not satisfy ℓ, we conclude that $\ell \in \{x, \bar{y}, \bar{z}\}$, which gives rise to three cases. As it will become clear soon, these three cases are the interesting cases in our analysis, whereas Cases 1.1, 1.2, and 2.1 can be ignored—by our analysis, of course, not by the algorithm.

Case 2.2 $\ell = x$. Dotted edges only set variables to 0, so they are of no help here. The algorithm changes $\alpha(D)$ to 101 by choosing the outgoing solid edge, and calls itself recursively:

$$\texttt{double-ball-search}(F, \alpha[D \to 101], s-1, t) \ .$$

Case 2.3 $\ell = \bar{y}$. We have several possibilities. A satisfying assignment that satisfies $\bar{y}$ could set D to 101, 001, 100, or 000. We do not want to cause four recursive calls. Note that if a satisfying assignment α^* sets D to 101, 001, or 100, then changing $\alpha(D)$ from 011 to 101 decreases $d(\alpha, \alpha^*)$ by 1 in any case. If $\alpha^*(D) = 000$, then changing $\alpha(D)$ to 000 decreases $\mathrm{cost}(\alpha, \alpha^*)$ by 2. Therefore the algorithm calls itself twice:

$$\texttt{double-ball-search}(F, \alpha[D \to 101], s-1, t)$$
$$\texttt{double-ball-search}(F, \alpha[D \to 000], s, t-2)$$

Case 2.4 $\ell = \bar{z}$. Observe that $\alpha^*(D)$ is either 110, 010, 100, or 000. If $\alpha^*(D)$ is 110 or 100, then changing $\alpha(D)$ from 011 to 110 decreases $d(\alpha, \alpha^*)$ by 2. If $\alpha^*(C)$ is 010 or 000, changing $\alpha(D)$ from 011 to 010 decreases $\mathrm{cost}(\alpha, \alpha^*)$ by 1. The algorithm thus calls itself twice:

$$\texttt{double-ball-search}(F, \alpha[D \to 110], s-2, t)$$
$$\texttt{double-ball-search}(F, \alpha[D \to 010], s, t-1)$$

For the analysis of the running time, we can ignore Cases 1.1, 1.2, and 2.1, since by Cases 2.2–2.4. Let $L(s,t)$ be the worst-case number of leaves in a recursion tree of $\texttt{double-ball-search}(F, \alpha, s, t)$.

Proposition 8. *If $s < 0$ or $t < 0$, then $L(s,t) = 1$. Otherwise,*

$$L(s,t) \leq L(s-1,t) + 2\max\begin{pmatrix} L(s-1,t) + L(s,t-2), \\ L(s-2,t) + L(s,t-1) \end{pmatrix} .$$

Proof. The proof works by induction. If $s < 0$ or $t < 0$, then clearly there is no assignment α^* with $d(\alpha, \alpha^*) \leq s$ and $\mathrm{cost}(\alpha, \alpha^*) \leq t$, and the algorithm simply returns failure.

If $s, t \geq 0$, let $C \in F \setminus \mathrm{Neg}(F)$ be a clause that α does not satisfy. In the worst case, $|C| = 3$ and consists of one positive and two negative literals. There are three cases: (i) the negative literals can be both of type $\bar{y}$ (Case 2.3), (ii) both of type $\bar{z}$ (Case 2.4), or (iii) one of type $\bar{y}$ and one of $\bar{z}$. If (i) holds, then we can bound $L(s,t)$ by

$$3L(s-1,t) + 2L(s,t-2) \ . \tag{4}$$

If (ii) holds, we can bound $L(s,t)$ by

$$L(s-1,t) + 2L(s-2,t) + 2L(s,t-1) \ . \tag{5}$$

Finally, if (iii) holds, we bound $L(s,t)$ by

$$L(s-1,t) + (L(s-1,t) + L(s,t-2)) + (L(s-2,t) + L(s,t-1)) \ ,$$

but one easily verifies that this is bounded from above by either (4) or (5). $\square$

Lemma 9. *Let $a, b \geq 1$ be such that*

$$ab^2 \geq b^2 + 2a \tag{6}$$

and

$$a^2 b \geq ab + 2a^2 + 2b \ . \tag{7}$$

Then $L(s,t) \in O(a^s b^t)$.

Proof. We use induction to show that $L(s,t) \leq C a^s b^t$ for some sufficiently large constant C. For the base case, i.e., $s < 0$ or $t < 0$, choose a constant C large enough that $1 = L(s,t) \leq C a^s b^t$. If $s, t \geq 0$, then conditions (6) and (7) guarantee that the induction goes through when one uses the bound from Proposition 8. $\square$

Combining `double-ball-search` with Covering Codes

Our overall algorithm works follows: If $\mathrm{Neg}(F)$ consists of at most r pairwise disjoint negative 3-clauses, it constructs a covering code $\mathcal{C}$ of radius s for the set of exact assignments. In other words, $\mathcal{C}$ is such that for every exact assignment β, there is some exact assignment $\alpha \in C$ with $d(\alpha, \beta) \leq s$. Here, s is some natural number to be determined later. It then calls

$$\texttt{double-ball-search}(F, \alpha, s, m)$$

for each $\alpha \in \mathcal{C}$, where $m := r - |\mathrm{Neg}(F)|$ is our "surplus budget". If no run of `double-ball-search` finds a satisfying assignment, it concludes that $B_r(1, \ldots, 1)$ contains no satisfying assignment, and returns failure. The overall running time of this business is

$$|\mathcal{C}| a^s b^{r-m} \mathrm{poly}(n) \ . \tag{8}$$

The following lemma is from Scheder [7], adapted to our current terminology.

Lemma 10 ([7])**.** *For every $x > 0$, there is some $s \in \{0, 1, \ldots, 2m\}$ and a covering code $\mathcal{C}$ of radius s for the set of exact assignments of size*

$$|\mathcal{C}| \leq \frac{3^m x^s}{(1+x+x^2)^m} \mathrm{poly}(m) \ .$$

Furthermore, one can deterministically construct $\mathcal{C}$ in time $O(|\mathcal{C}|)$.

Combining (8) with the lemma and setting $x := 1/a$ in the lemma, we see that the running time of the algorithm is at most

$$\frac{3^m x^s a^s}{(1+x+x^2)^m} b^{r-m} \mathrm{poly}(n) = \left(\frac{3a^2}{a^2+a+1} \right)^m b^{r-m} \mathrm{poly}(n) \ . \tag{9}$$

We still can choose a and b, as long as they satisfy (6) and (7). We try the following: We guess (with hindsight, we know) that (5) dominates the running time, thus we try to satisfy (7) with equality. Furthermore, we want to get rid of the parameter m in (9),

which depends on the formula F and over which we do not have control. In other words, we want to choose a and b such that

$$\begin{aligned} a^2 b &= ab + a^2 + 2b \\ b &= \frac{3a^2}{a^2 + a + 1} \,. \end{aligned}$$

One checks that $a = (5 + \sqrt{57})/2$ and $b = (5 + \sqrt{57})^2/(4(8 + \sqrt{57}))$ will do, and also satisfy (6). With these numbers, (9) boils down to $O^*(b^r) \leq O^*(2.533^r)$. This finishes the proof of Lemma 4.

We observe that our algorithm solves the case where $\text{Neg}(F)$ consists of pairwise disjoint negative 3-clauses more efficiently, in $O^*(2.533^r)$, than the case where F contains overlapping negative clauses, in which there is a branching leading to a running time of $O^*(2.562^r)$. This is qualitatively different from all previous approaches to improving local search (Dantsin et al. [2], Brueggeman and Kern [1] and Scheder [6]): In those approaches, pairwise disjoint negative clauses constitute the worst case, and the case where F contains intersecting clauses is always the easy case handled at the very beginning. Now the picture has changed: A further improvement will have to work on that case, too.

We are ready to prove Theorem 2.

Proof of Theorem 2. The algorithm outlined above solves Ball-3-SAT in time $O^*(a^r)$, where $a \approx 2.562$, with the worst case being that F contains negative clauses that intersect in one literal. Combining this with a standard construction of covering codes, we conclude that 3-SAT can be solved in time

$$\left(\frac{2a}{a+1}\right)^n \text{poly}(n) \leq O^*\left(1.439^n\right) \,,$$

which finishes the proof. □

References

[1] T. Brueggemann and W. Kern. An improved deterministic local search algorithm for 3-sat. *Theor. Comput. Sci.*, 329(1-3):303–313, 2004.

[2] E. Dantsin, A. Goerdt, E. A. Hirsch, R. Kannan, J. Kleinberg, C. Papadimitriou, O. Raghavan, and U. Schöning. A deterministic $(2 - 2/(k+1))^n$ algorithm for k-SAT based on local search. In *Theoretical Computer Science 289*, pages 69–83, 2002.

[3] R. Paturi, P. Pudlák, M. E. Saks, and F. Zane. An improved exponential-time algorithm for k-SAT. *J. ACM*, 52(3):337–364, 2005.

[4] R. Paturi, P. Pudlák, and F. Zane. Satisfiability coding lemma. *Chicago J. Theoret. Comput. Sci.*, pages Article 11, 19 pp. (electronic), 1999.

[5] D. Rolf. Improved bound for the PPSZ/Schöning-algorithm for 3-SAT. *Electronic Colloquium on Computational Complexity (ECCC)*, (159), 2005.

[6] D. Scheder. Guided search and a faster deterministic algorithm for 3-SAT. In *Proc. of the 8th Latin American Symposium on Theoretical Informatics (LATIN'08), Lecture Notes In Computer Science, Vol. 4957*, pages 60–71, 2008.

[7] D. Scheder. Using a skewed hamming distance to speed up deterministic local search. *CoRR*, abs/1005.4874, 2010.

[8] U. Schöning. A probabilistic algorithm for k-SAT and constraint satisfaction problems. In *FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, page 410, Washington, DC, USA, 1999. IEEE Computer Society.